Project Write-Up

Dahlia Radif

December 13, 2018

1 Introduction

In this project, we use the conjugate gradient method to solve the 2D heat equation on a cross section of a pipe. Hot fluid is transferred in the pipe, and a series of cold air jets are evenly distributed along the pipe wall. In this document, the methodology used to obtain the mean temperature and the pseudocolor plots will be discussed.

2 Implementation of CG Solver Algorithm

In this section, we will discuss the C++ implementation of the CG Solver, explaining the different classes used and how they interact in order to find the solution vector of temperatures within the discretised pipe wall.

2.1 C++ Files and Classes

Five cpp files are used in the C++ portion of the code in order to form and solve the linear system modelling the temperatures within the pipe. These are `matvecops.cpp`, `heat.cpp`, `sparse.cpp`, `COO2CSR.cpp`, and `CGSolver.cpp`. Their respective helper files are also included. The class `HeatEquation2D` contains a function, `SetUp` that forms the system of linear equations into a COO format matrix,and converts it to CSR format using the relevant method in `COO2CSR.cpp`. The second function is `Solve`, which calls the method `CGSolver` in order to obtain the number of iterations required for the solution vector to converge. A more in-depth explanation of the conjugate gradient algorithm and its implementation is provided in Sections 2.3 and 2.4.

2.2 OOP Matrix Design and Implementation

In order to form the matrix of linear equations, it is important to understand the geometry of the problem[1]. We imagine taking a cross section of the pipe, and transforming it into a discrete Cartesian grid. The top part of the pipe is the top row of the grid, corresponding to the hot isothermal boundary, T_h, while the bottom row of the grid corresponds to the cold isothermal boundary, $T_c(x)$. This has an inverted Gaussian temperature distribution[2]. We also assume periodic boundary conditions, so that the first and last column are coincident, so we do not need to include the last column in our matrix. Every point within this grid corresponds to a linear equation that is used to obtain the temperature at that particular point. In other words, every point within the grid corresponds to a row in the matrix. If we assume that we are at position (i,j) in the grid (numbering starts from the bottom left), then the corresponding row in the matrix is `j + ((i - 1) * (nj-1))`, where n_j is the number of columns in the Cartesian grid (including both periodic boundaries).

In the unknown vector x and the right side vector b, we use `reserve` to reserve enough space in memory for the total number of elements that will be appended to them. Note that we initially start with a vector of ones for the intial solution. The use of if-else statements is to ensure that we properly populate each row of the sparse matrix[3]; for those points near the the top or bottom boundary conditions, we need to account for the fact that the solution vector will contain the respective temperature value, rather than zero. Similarly, we implement a wrap-around methodology for those grid points lying on the periodic boundary. In the `HeatEquation2D` class, a function called `lowerIsothermalBoundary` calculates the value of $T_c(x)$. After iterating through all the grid points and appending the respective values in each row (note that we form the system $(-A)u = -b$ because the original system is negative definite[4]), we convert the matrix from COO to CSR format, and check that the size of the row pointer vector excluding the last entry is equal to the size of the matrix; if not, then the system has not been formed correctly and an error is created. In this class, we create an instance of the sparse matrix class which allows us to use its methods to help solve the system.

The `SparseMatrix` class contains general attributes and methods for sparse matrices; for example, it is not mandatory for the matrix to be square (but note that when it is used to form this particular linear system, the matrix is always square). The attributes are the three vectors of a CSR matrix, the number of rows and columns of the matrix (in dense form), and a boolean variable CSR indicating whether the matrix has been converted into that format yet. We also make `HeatEquation2D` a friend class in order to access the private attributes. The method `Resize` is implemented but never used, and simply sets the attributes nrows and ncols equal to the dimension of the linear system. The method `AddEntry` adds an entry to the matrix in COO format, and the method `ConvertToCSR` calls the conversion method from `COO2CSR.cpp`. `GetSize` returns the size of the matrix when it is in CSR format. The two remaining functions, `MulVec` and `A_norm` call functions from `matvecops.cpp` (see Section 2.4) and implement the respective methodology using the CSR matrix attributes.

2.3 Conjugate Gradient Method

The pseudo-code for the conjugate gradient method[5] is shown in Algorithm 1.

2.4 Implementation of CG Solver Algorithm

In order to implement the CG solver, we use an instance of the `SparseMatrix` class as well as matrix-vector methods from `matvecops.cpp`, in order to eliminate redundant code and improve the clarity of the algorithm. The functions in this file are `twoNorm`, `dotProduct`, `constVecMultiply`, `matVecMultiply`, `A_norm`, `vectorSubtraction`, and `vectorAddition`. The sparse matrix instance allows us to use the matrix-vector multiplication method and A-norm method implemented in the class. In many of the functions, casting to `unsigned int` is necessary when compiling using `-Wconversion`.

The final part of the CG solver function determines which value of n to return; if we converge to a solution within the specified number of iterations, n is returned, otherwise we return -1. The solution vector is written to a suitably named `.txt` file every 10 iterations, including the first and last solution vectors.

In addition, many of these functions implement dimension checking, namely in `dotProduct`, `matVecMultiply` , `vectorSubtraction`, and `vectorAddition`. This is important to ensure that we do not allow, for example, the dot product computation between two vectors of different

Algorithm 1 Conjugate Gradient Algorithm.

$u_0 \leftarrow initial\ value\ for\ u_0$
$nitermax \leftarrow initial\ value\ for\ nitermax$
$r_0 = b - Au_0$
$L2normr0 = \|r_0\|_2$
$p_0 \leftarrow r_0$
$niter \leftarrow 0$
while $niter < nitermax$ **do**
  $niter \leftarrow niter + 1$
  $alpha_n \leftarrow r_n^T r_n / p_n^T A p_n$
  $u_{n+1} \leftarrow u_n + alpha_n p_n$
  $r_{n+1} \leftarrow r_n - alpha_n A p_n$
  $L2normr = \|r_{n+1}\|_2$
  if $L2normr/L2normr0 < threshold$ **then**
    break
  else
    $beta_n \leftarrow r_{n+1}^T r_{n+1} / r_n^T r_n$
    $p_{n+1} \leftarrow r_{n+1} + beta_n p_n$
  end if
end while

sizes. When the dimension check fails, these functions provide an error message to the user, and then exit the program.

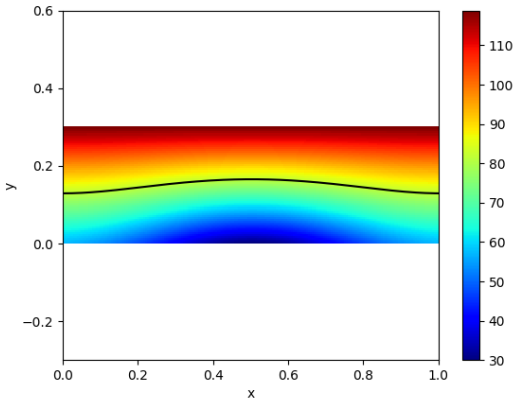
3 Users' Guide

In this section, we will discuss how to implement and compile the code. In C++, we compile the code using the `makefile`. Calling `make` compiles the code, creating object files and the main executable, and then we call main with the correct number of inputs; these are the input file, containing the length, width, and h value of the pipe, as well as T_c and T_h, and the prefix name of the solution file, e.g. 'solution'. For example, the input to the terminal could be `./main input2.txt solution` and the output, if convergence occurs, would be `SUCCESS: CG solver converged in xx iterations.` For `input2.txt`, it takes 157 iterations.

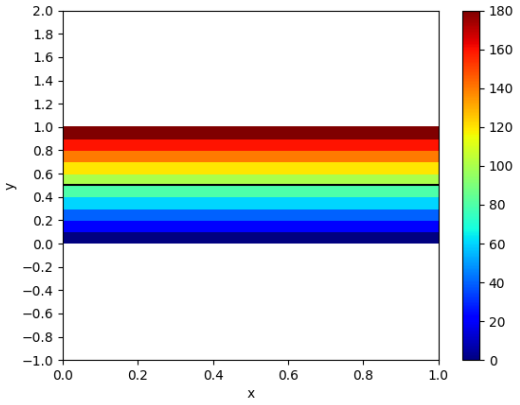
The second file is a Python processing script that takes three inputs and returns a pseudocolor plot of the thermal distribution within the pipe. For example, a terminal command should be something similar to `python3 postprocess.py input2.txt solution157.txt imagefile.png`. Inputting the final solution vector is crucial for ensuring the most accurate psuedocolor plot. See Figure 1 for the pseudcolor plots for `input2.txt` and `input0.txt`. The Python script uses linear interpolation in order to calculate the mean isoline, and uses the `pcolor` function in the `matplotlib` library in order to generate the plot. The output to the terminal will be the mean global temperature, and the plot will be written to the third terminal input, namely `imagefile.png`.

Finally, the Python script `bonus.py` generates an animation that shows the temperature distribution development during the CG algorithm. It takes two inputs; the original input file, e.g. `input2.txt` and a directory, which contains all of the vectors solutions after every 10 iterations, including the first and last. For example, a terminal input could be `python3 bonus.py input2.txt input2files`, where `input2files` contains the txt files of the vectors formed during the convergence

of that input file. This function will then generate a pseudocolor plot for each vector, and display an animation showing the way in which the temperature changes during the implementation of the conjugate gradient algorithm.



(a) Pseudocolor plot with mean temperature isoline for input2.txt



(b) Pseudocolor plot with mean temperature isoline for input0.txt

Figure 1: Example pseudocolor plots with mean isolines

References

[1] CME 211. "CME 211 Final Project." 2018. PDF file, pages 1-3.

[2] CME 211. "CME 211 Final Project." 2018. PDF file, page 4.

[3] CME 211. "CME 211 Final Project." 2018. PDF file, page 3.

[4] CME 211. "CME 211 Final Project." 2018. PDF file, page 3.

[5] CME 211. "CME 211 HW 4." 2018. PDF file, page 2.